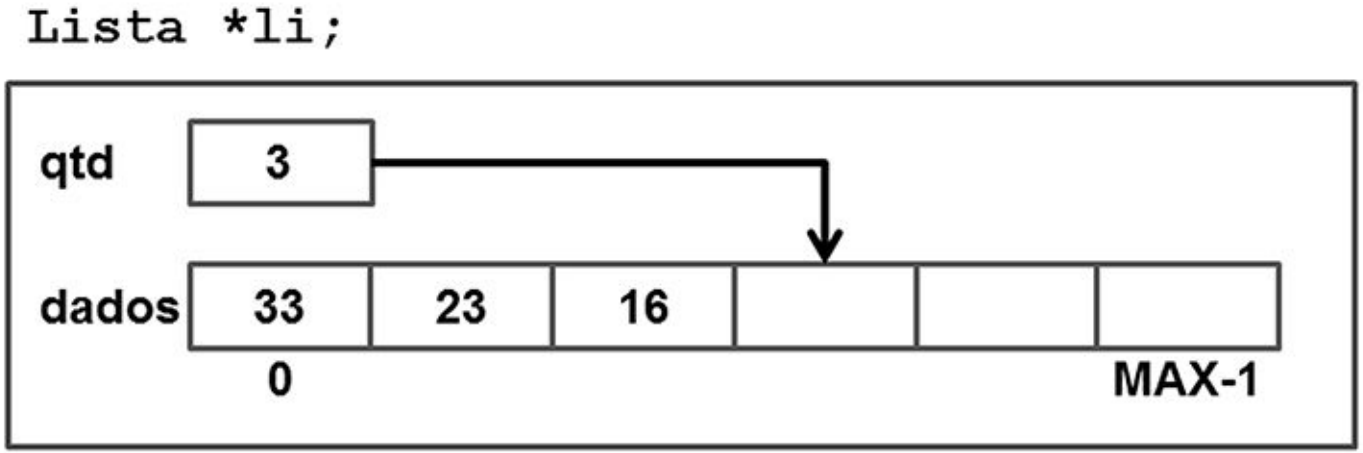
Listas Encadeadas/Ligadas

Sergio Canuto
sergio.canuto@ifg.edu.br

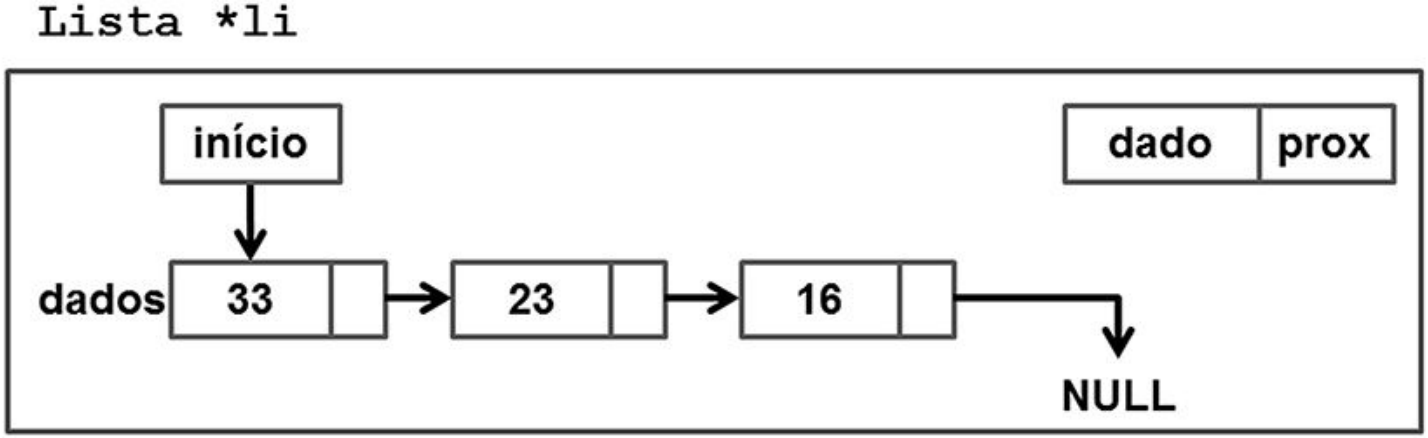
Relembrando...

- **LISTA SEQUENCIAL ESTÁTICA**
  - Uma **lista sequencial estática** ou **lista linear estática** é uma lista definida utilizando alocação estática e acesso sequencial dos elementos



LISTA DINÂMICA ENCADEADA (Lista Ligada)

- Uma **lista dinâmica encadeada** é uma lista definida utilizando alocação dinâmica e acesso encadeado dos elementos. Cada elemento contém:
  - um campo de **dado**, utilizado para armazenar a informação.
  - um campo **prox**, ponteiro que indica o próximo elemento na lista.
  - Usa um ponteiro especial para o primeiro elemento da lista e uma indicação para o final da lista.



Lista Encadeada/ligada

Arquivo ListaDinEncad.h

```
01  struct aluno{
02      int matricula;
03      char nome[30];
04      float n1,n2,n3;
05  };
06  typedef struct elemento* Lista;
07
08  Lista* cria_lista();
09  void libera_lista(Lista* li);
10  int insere_lista_final(Lista* li, struct aluno al);
11  int insere_lista_inicio(Lista* li, struct aluno al);
12  int insere_lista_ordenada(Lista* li, struct aluno al);
13  int remove_lista(Lista* li, int mat);
14  int remove_lista_inicio(Lista* li);
15  int remove_lista_final(Lista* li);
16  int tamanho_lista(Lista* li);
17  int lista_vazia(Lista* li);
18  int lista_cheia(Lista* li);
19  int busca_lista_mat(Lista* li, int mat, struct aluno *al);
20  int busca_lista_pos(Lista* li, int pos, struct aluno *al);
```

- **não precisa definir a quantidade MAX de elementos**
- **a Lista não é mais um array de dados, mas um ponteiro para elemento.**

Lista Encadeada

Arquivo ListaDinEncad.c

```
01  #include <stdio.h>
02  #include <stdlib.h>
03  #include "ListaDinEncad.h" //inclui os protótipos
04  //Definição do tipo lista
05  struct elemento{
06      struct aluno dados;
07      struct elemento *prox;
08  };
09  typedef struct elemento Elem;
```

Criando uma lista

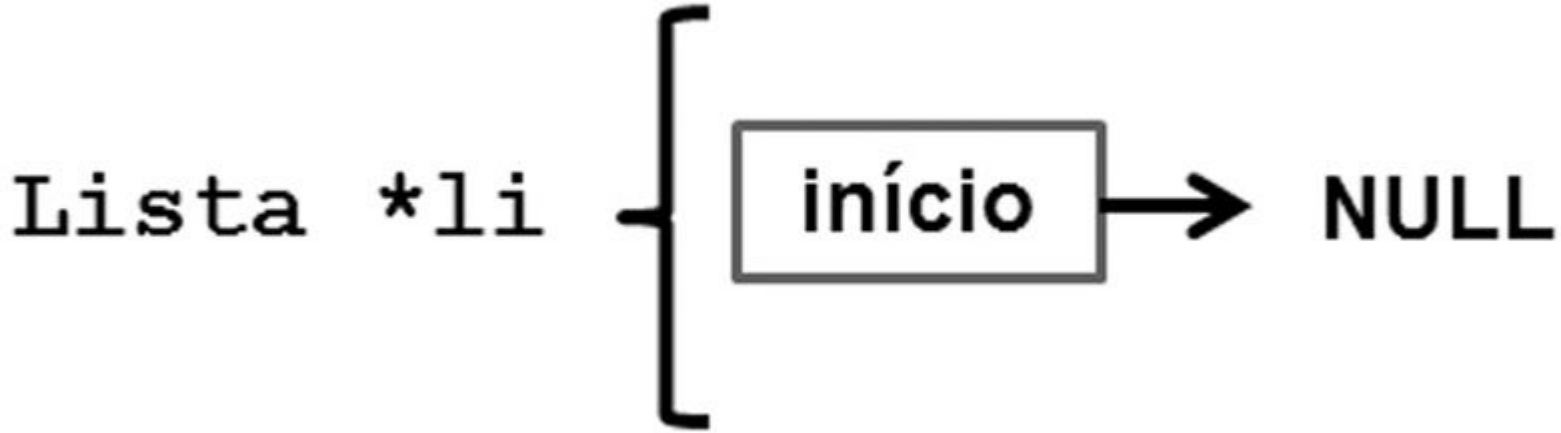
- Para utilizar uma lista em seu programa, a primeira coisa a fazer é criar uma lista vazia.
- Alocar de uma área de memória para armazenar o endereço do início da lista (linha 2), que é um ponteiro para ponteiro.
- Esta área de memória corresponde à memória necessária para armazenar o endereço de um elemento da lista, **sizeof(Lista)** ou **sizeof(struct elemento*)**.
- Em seguida, a função inicializa o conteúdo desse **ponteiro para ponteiro** com a constante **NULL**.
- Esta constante é utilizada em uma **lista dinâmica encadeada** para indicar que não existe nenhum elemento alocado após o atual.
- Como o início da lista aponta para tal constante, isso significa que a lista está vazia.

Criando uma lista

```
01  Lista* cria_lista(){
02      Lista* li = (Lista*) malloc(sizeof(Lista));
03      if(li != NULL)
04          *li = NULL;
05      return li;
06  }
```

Criando uma lista

- Conteúdo do nosso ponteiro **Lista* li** após a chamada da função que cria a lista.



Destruindo uma lista

- Destruir uma lista que utilize alocação dinâmica, e seja encadeada, **não é uma tarefa tão simples** quanto destruir uma **lista sequencial estática**.
  - Inicialmente, verificamos se a lista é válida (linha 2). Em seguida, percorremos a lista até que o conteúdo do seu início (***li**) seja diferente de **NULL**, o final da lista. Enquanto não chegarmos ao final da lista, iremos liberar a memória do elemento que se encontra atualmente no início da lista e avançar para o próximo (linhas 5-7). Terminado o processo, liberamos a memória alocada para o início da lista (linha 9).

Destruindo uma lista

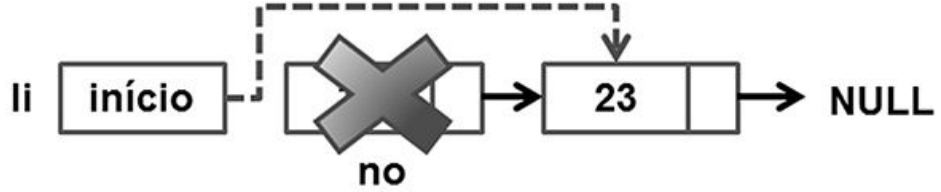
```
01  void libera_lista(Lista* li){
02      if(li != NULL){
03          Elem* no;
04          while((*li) != NULL){
05              no = *li;
06              *li = (*li)->prox;
07              free(no);
08          }
09          free(li);
10      }
11  }
```

Destruindo uma lista

Lista inicial

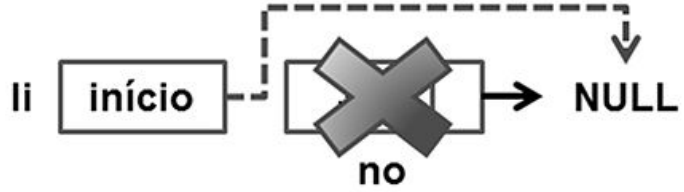
Passo 1:

```
no = *li;
*li = (*li)->prox;
free(no);
```



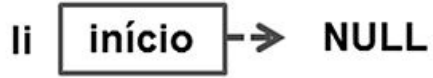
Passo 2:

```
no = *li;
*li = (*li)->prox;
free(no);
```



Fim:

```
no == NULL
```



Informações básicas sobre a lista

- As operações de inserção, remoção e busca são consideradas as principais de uma lista.
- Apesar disso, para realizar estas operações é necessário ter em mãos outras informações mais básicas sobre a lista, como:
  - Tamanho da Lista
  - Lista Cheia?
  - Lista Vazia?

Informações básicas sobre a lista - Tamanho da lista

- Inicialmente, verificamos se a lista é válida (linha 2). Em seguida, criamos um contador iniciado em **ZERO** (linha 4) e um elemento auxiliar (**no**) apontado para o primeiro elemento da lista (linha 5). Então, percorremos a lista até que o valor de **no** seja diferente de **NULL** (linhas 6-9). Terminado o processo, retornamos o valor da variável **cont** (linha 10).

Tamanho da lista

```
01  int tamanho_lista(Lista* li){
02      if(li == NULL)
03          return 0;
04      int cont = 0;
05      Elem* no = *li;
06      while(no != NULL){
07          cont++;
08          no = no->prox;
09      }
10      return cont;
11  }
```

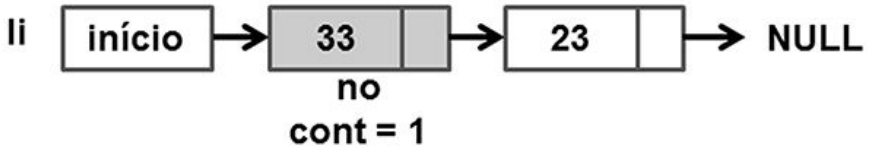
Informações básicas sobre a lista - Tamanho da lista

Lista inicial:

```
cont = 0;
no = *li;
```

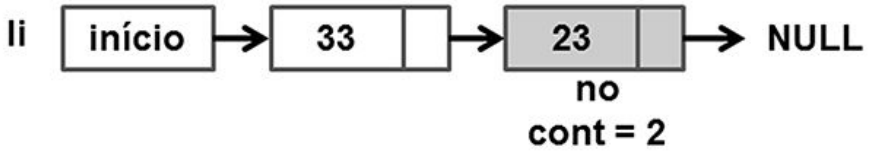
Passo 1:

```
cont++;
no = no->prox;
```



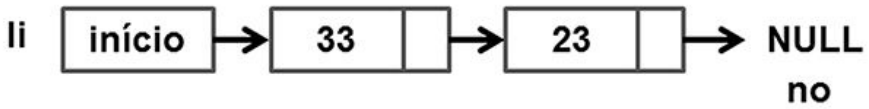
Passo 2:

```
cont++;
no = no->prox;
```



Fim:

```
no == NULL
```



Informações básicas sobre a lista - Lista Cheia

- Sempre retorna Zero! (lista nunca cheia)

Exemplo

```
01  int lista_cheia(Lista* li){
02      return 0;
03  }
```

Informações básicas sobre a lista - Lista Vazia

- Se a lista foi criada com sucesso (linha 3) ou se seu início aponta para um elemento **NULL** (linha 5), a lista é vazia.
- Caso contrário, irá retornar o valor **ZERO** (linha 6).

Retornando se a lista está vazia

```
01  int lista_vazia(Lista* li){
02      if(li == NULL)
03          return 1;
04      if(*li == NULL)
05          return 1;
06      return 0;
07  }
```

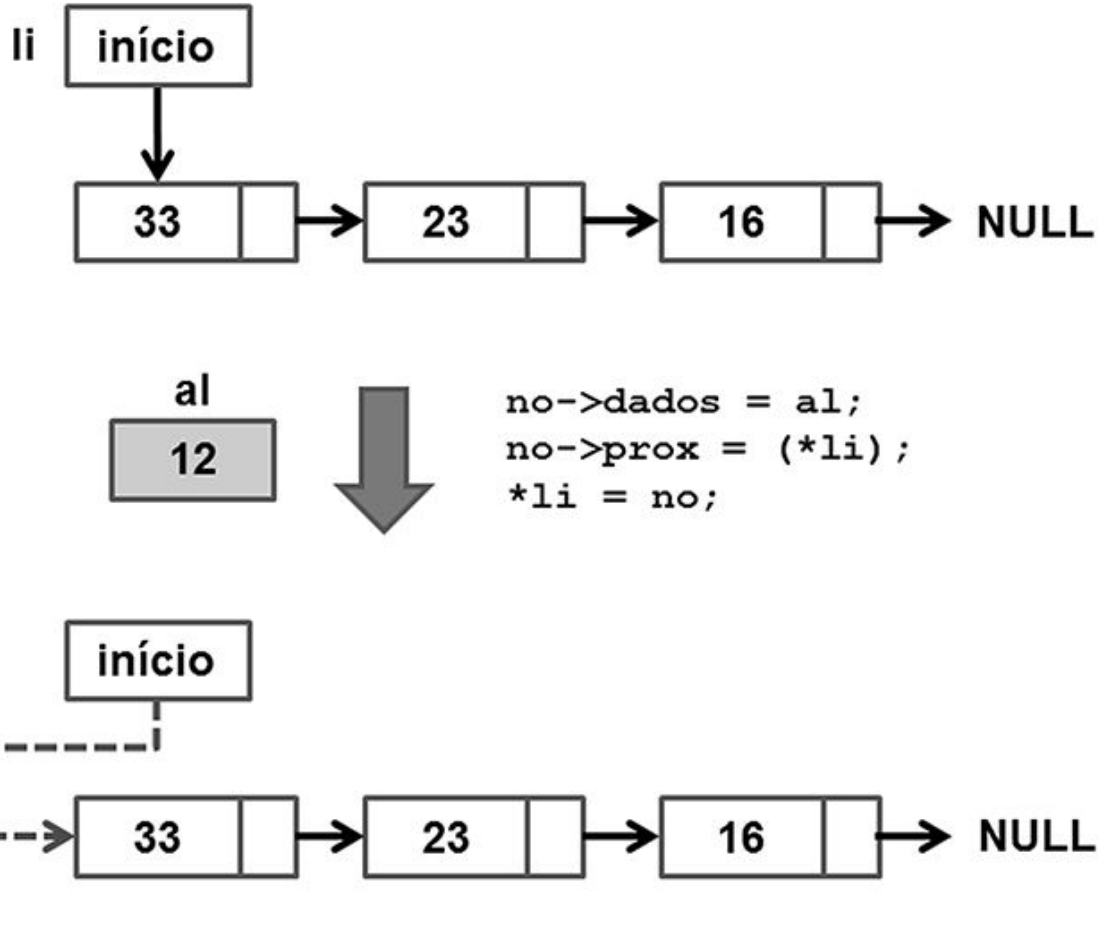
Inserindo um elemento na lista - Início

- Linhas 2 a 8 verificam se a inserção é possível. Como se trata de uma inserção no início, temos que fazer nosso elemento apontar para o início da lista, ***li** (linha 9). Assim, o elemento **no** passa a ser o início da lista, enquanto o antigo início passa a ser o próximo elemento da lista. Por fim, mudamos o conteúdo do "início" da lista (***li**) para que ele passe a ser o nosso elemento **no** e retornamos o valor **UM** (linhas 10 e 11), indicando sucesso.

Inserindo um elemento no início da lista

```
01  int insere_lista_inicio(Lista* li, struct aluno al){
02      if(li == NULL)
03          return 0;
04      Elem* no;
05      no = (Elem*) malloc(sizeof(Elem));
06      if(no == NULL)
07          return 0;
08      no->dados = al;
09      no->prox = (*li);
10      *li = no;
11      return 1;
12  }
```

Inserindo um elemento na lista - Início



Inserindo um elemento na lista - final

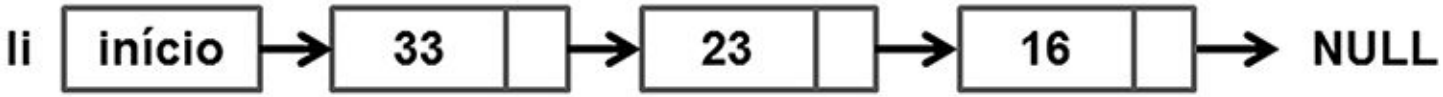
- Inserir um elemento no final de uma **lista dinâmica encadeada** é uma tarefa um tanto trabalhosa:

Inserindo um elemento no final da lista

```
01  int insere_lista_final(Lista* li, struct aluno al){
02      if(li == NULL)
03          return 0;
04      Elem *no;
05      no = (Elem*) malloc(sizeof(Elem));
06      if(no == NULL)
07          return 0;
08      no->dados = al;
09      no->prox = NULL;
10      if((*li) == NULL){//lista vazia: insere início
11          *li = no;
12      }else{
13          Elem *aux;
14          aux = *li;
15          while(aux->prox != NULL){
16              aux = aux->prox;
17          }
18          aux->prox = no;
19      }
20      return 1;
21  }
```

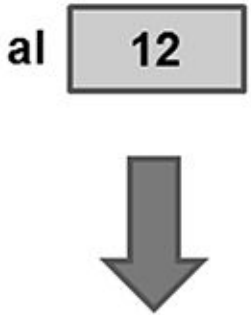
Inserindo um elemento na lista - final

- Inserir um elemento no final de uma **lista dinâmica encadeada** é uma tarefa um tanto trabalhosa:



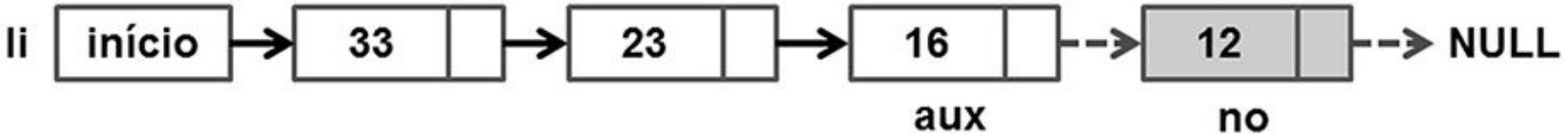
Busca onde Inserir:

```
aux = *li;
while(aux->prox != NULL){
    aux = aux->prox;
}
```



Insere depois de "aux":

```
no->dados = al;
no->prox = NULL;
aux->prox = no;
```



Removendo um elemento na lista - início

- Remover um elemento do início de uma **lista dinâmica encadeada** é uma tarefa bastante simples.

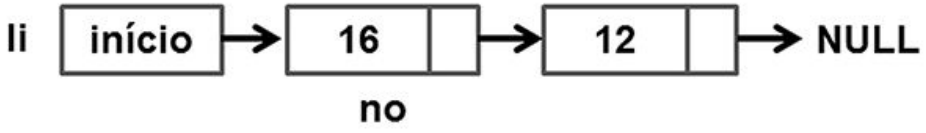
Removendo um elemento do início da lista

```
01  int remove_lista_inicio(Lista* li){
02      if(li == NULL)
03          return 0;
04      if((*li) == NULL)//lista vazia
05          return 0;
06
07      Elem *no = *li;
08      *li = no->prox;
09      free(no);
10      return 1;
11  }
```

Removendo um elemento na lista - início

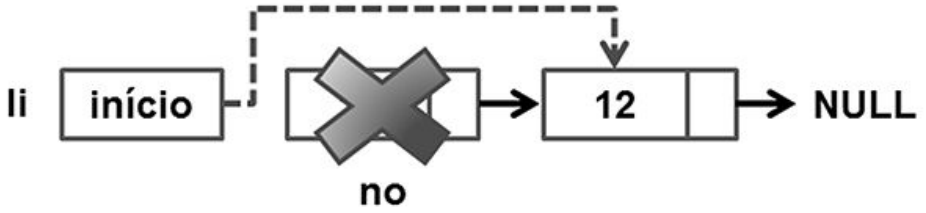
- Remover um elemento do início de uma **lista dinâmica encadeada** é uma tarefa bastante simples.

Lista inicial

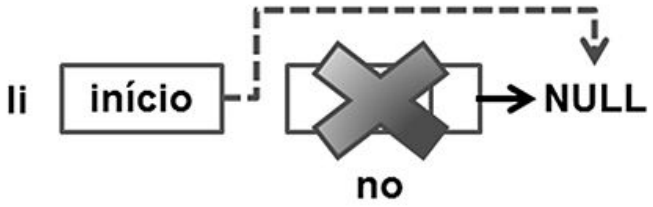



```
*li = no->prox;
free(no);
```

Se a lista possui mais de um elemento, o início aponta para o segundo



Se a lista possui um único elemento, ela fica vazia

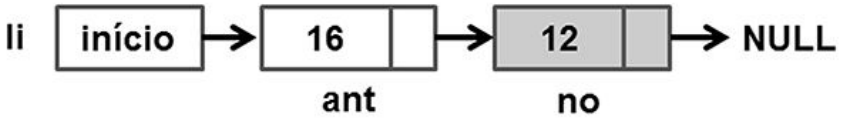


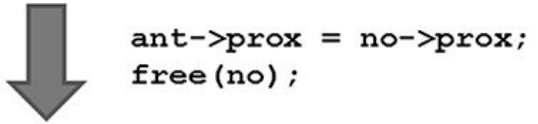
Removendo um elemento na lista - final

- Remover um elemento do final de uma **lista dinâmica encadeada** é uma tarefa um tanto trabalhosa.

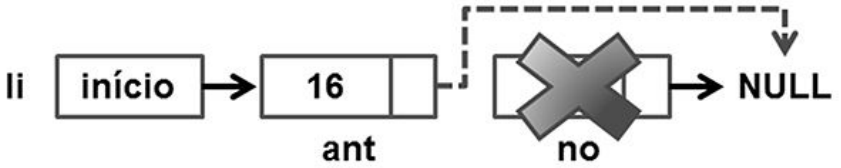
Lista inicial
Busca o último elemento:

```
no = *li;
while(no->prox != NULL){
    ant = no;
    no = no->prox;
}
```



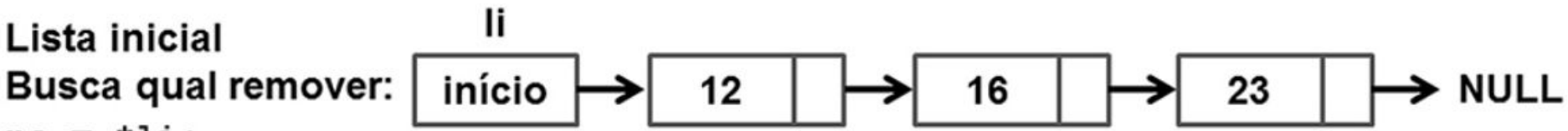


Se a lista possui mais de um elemento, ant aponta para NULL



Removendo um elemento específico na lista

- Remover um elemento específico de uma **lista dinâmica encadeada** é uma tarefa trabalhosa.




```
no = *li;
while(no != NULL && no->dados.matricula != mat){
    ant = no;
    no = no->prox;
}
```

Buscando elementos na lista

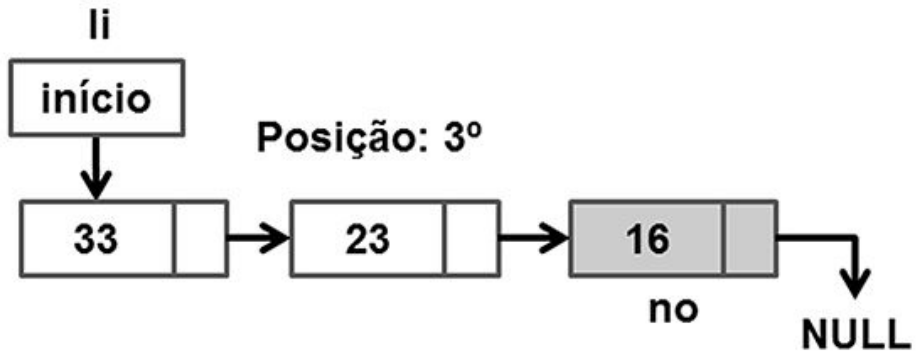
- Buscar um elemento específico de uma **lista dinâmica encadeada** é uma tarefa tão trabalhosa por posição quanto por conteúdo:
  - Pior caso, percorre todos os elementos.

Busca pela posição do elemento

```
no = *li;
int i = 1;
while(no != NULL && i < pos){
   no = no->prox;
   i++;
}
```

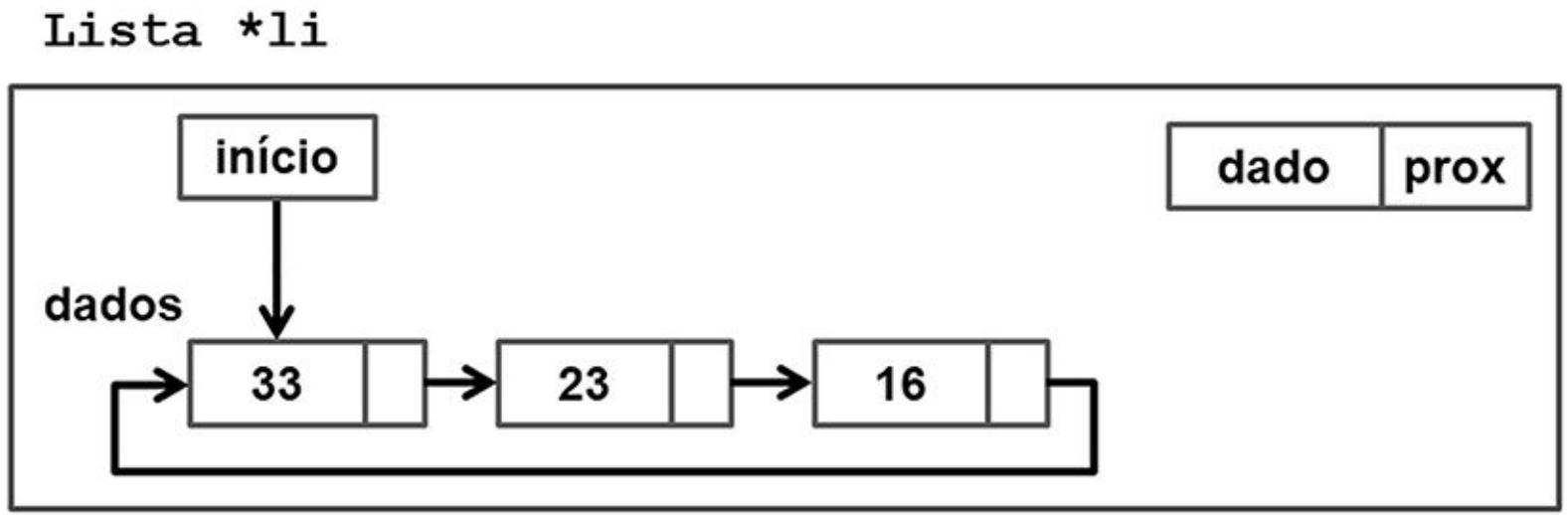
Verifica se a posição foi encontrada e a retorna

```
if(no == NULL) return 0;
else{
   *al = no->dados;
   return 1;
}
```



LISTA DINÂMICA ENCADEADA CIRCULAR

- Em uma **lista dinâmica encadeada**, após o último elemento não existe nenhum novo elemento alocado, de modo que o último elemento da lista aponta para **NULL**. Já em uma **lista dinâmica encadeada circular**, o último elemento tem como sucessor o primeiro elemento da lista.
- Parece não ter fim: nunca chegaremos a uma posição final a partir da qual não poderemos mais andar dentro da lista, porque depois do último elemento voltamos para o primeiro, como em um círculo.
- Uso: Há a necessidade de voltar ao primeiro elemento da lista depois de percorrê-la (ex.: escalonamento de processos)



LISTA DINÂMICA ENCADEADA CIRCULAR

- Mesmas definições da lista dinâmica encadeada (não circular) anterior.

Arquivo ListaDinEncadCirc.h

```
01  struct aluno{
02      int matricula;
03      char nome[30];
04      float n1,n2,n3;
05  };
06  typedef struct elemento* Lista;
07
08  Lista* cria_lista();
09  void libera_lista(Lista* li);
10  int busca_lista_pos(Lista* li, int pos, struct aluno *al);
11  int busca_lista_mat(Lista* li, int mat, struct aluno *al);
12  int insere_lista_final(Lista* li, struct aluno al);
13  int insere_lista_inicio(Lista* li, struct aluno al);
14  int insere_lista_ordenada(Lista* li, struct aluno al);
15  int remove_lista(Lista* li, int mat);
16  int remove_lista_inicio(Lista* li);
17  int remove_lista_final(Lista* li);
18  int tamanho_lista(Lista* li);
19  int lista_vazia(Lista* li);
20  int lista_cheia(Lista* li);
```

Arquivo ListaDinEncadCirc.c

```
01  #include <stdio.h>
02  #include <stdlib.h>
03  #include "ListaDinEncadCirc.h" //inclui os Protótipos
04  //Definição do tipo lista
05  struct elemento{
06      struct aluno dados;
07      struct elemento *prox;
08  };
09  typedef struct elemento Elem;
```

LISTA DINÂMICA ENCADEADA CIRCULAR

- Não há diferença entre a criação de uma lista dinâmica encadeada e sua versão circular:

Criando uma lista

```
01  Lista* cria_lista(){
02      Lista* li = (Lista*) malloc(sizeof(Lista));
03      if(li != NULL)
04          *li = NULL;
05      return li;
06  }
```

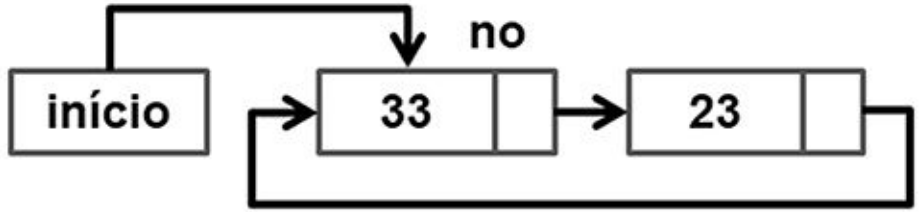
LISTA DINÂMICA ENCADEADA CIRCULAR - destruição

- Mas na destruição, percorremos todos elementos até chegar no início novamente (e não até encontrar um ponteiro null)

Destruindo uma lista

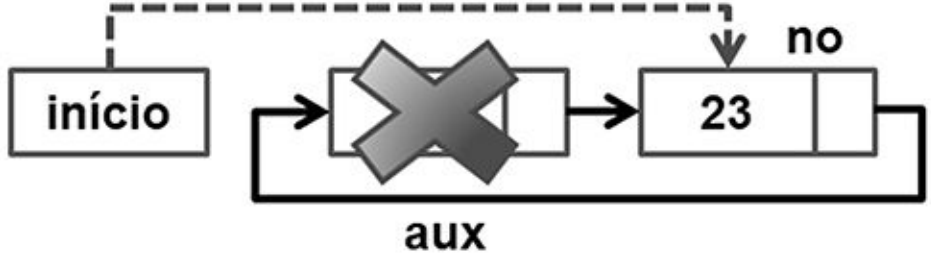
```
01  void libera_lista(Lista* li){
02      if(li != NULL && (*li) != NULL){
03          Elem *aux, *no = *li;
04          while((*li) != no->prox){
05              aux = no;
06              no = no->prox;
07              free(aux);
08          }
09          free(no);
10          free(li);
11      }
12  }
```

Lista inicial



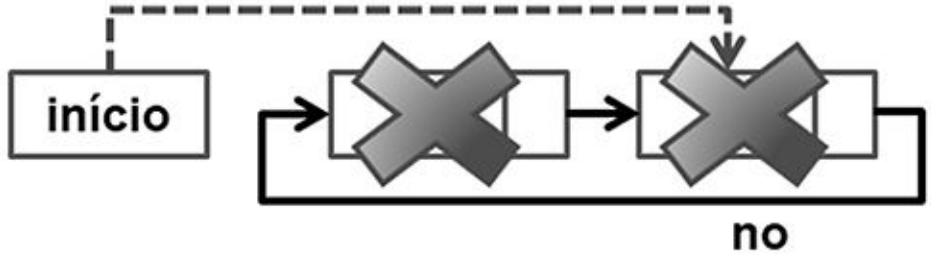
Passo 1:

```
aux = no;
no = no->prox;
free(aux);
```



Fim:

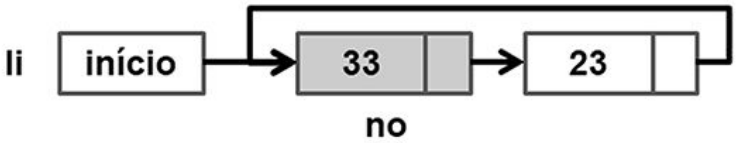
```
no->prox == *li

free(no);
```



LISTA DINÂMICA ENCADEADA CIRCULAR - tamanho

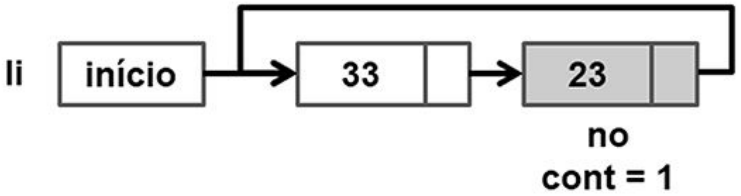
Lista inicial:

```
cont = 0;
no = *li;
```



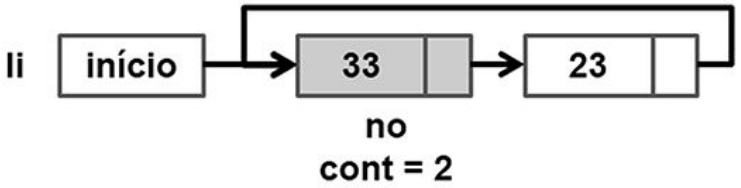
Passo 1:

```
cont++;
no = no->prox;
```



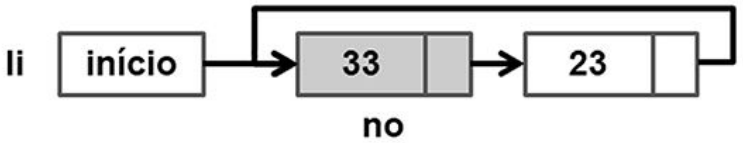
Passo 2:

```
cont++;
no = no->prox;
```



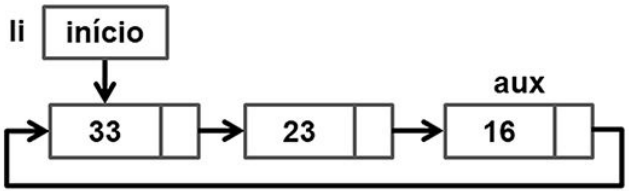
Fim:

```
no == *li
```



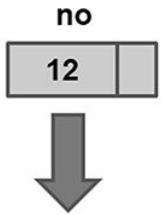
LISTA DINÂMICA ENCADEADA CIRCULAR - Inserindo no início da lista

- Inserir um elemento no início de uma **lista dinâmica encadeada circular** pode ser uma tarefa um tanto trabalhosa.
  - Encontrar o último da lista e mudar os ponteiros para o início
  - É possível melhorar?



Busca último elemento "aux":

```
aux = *li;
while(aux->prox != (*li)){
    aux = aux->prox;
}
```



Insere depois de "aux":

```
aux->prox = no;
no->prox = *li;
*li = no;
```

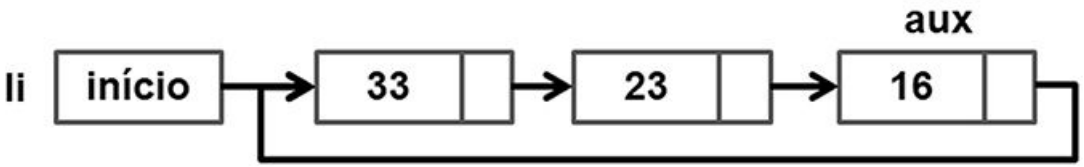
LISTA DINÂMICA ENCADEADA CIRCULAR - Inserindo no início da lista

- **Inserindo um elemento no início da lista**

```
01  int insere_lista_inicio(Lista* li, struct aluno al){
02      if(li == NULL)
03          return 0;
04      Elem *no = (Elem*) malloc(sizeof(Elem));
05      if(no == NULL)
06          return 0;
07      no->dados = al;
08      if((*li) == NULL){//lista vazia: insere início
09          *li = no;
10          no->prox = no;
11      }else{
12          Elem *aux = *li;
13          while(aux->prox != (*li)){
14              aux = aux->prox;
15          }
16          aux->prox = no;
17          no->prox = *li;
18          *li = no;
19      }
20      return 1;
21  }
```

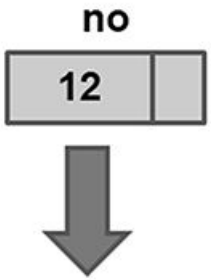
LISTA DINÂMICA ENCADEADA CIRCULAR - Inserindo no final da lista

- Inserir um elemento no final de uma **lista dinâmica encadeada circular** é tão trabalhosa quanto inserir no início:
  - Encontrar o último da lista e mudar os ponteiros para o início
  - É possível melhorar?



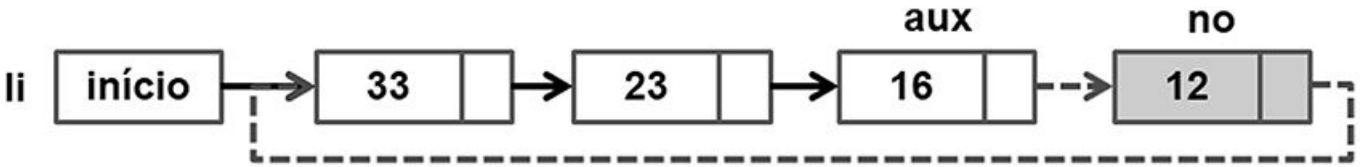
Busca último elemento "aux":

```
aux = *li;
while(aux->prox != (*li)){
    aux = aux->prox;
}
```



Insere depois de "aux":

```
aux->prox = no;
no->prox = *li;
```



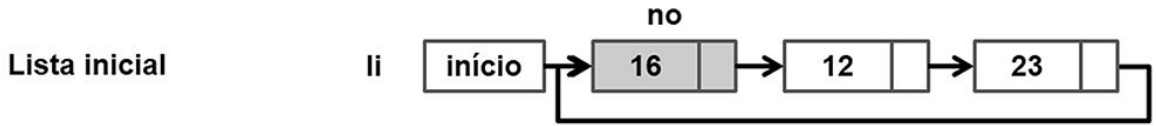
LISTA DINÂMICA ENCADEADA CIRCULAR - Inserindo no final da lista

- **Inserindo um elemento no final da lista**

```
01  int insere_lista_final(Lista* li, struct aluno al){
02      if(li == NULL)
03          return 0;
04      Elem *no = (Elem*) malloc(sizeof(Elem));
05      if(no == NULL)
06          return 0;
07      no->dados = al;
08      if((*li) == NULL){//lista vazia: insere início
09          *li = no;
10          no->prox = no;
11      }else{
12          Elem *aux = *li;
13          while(aux->prox != (*li)){
14              aux = aux->prox;
15          }
16          aux->prox = no;
17          no->prox = *li;
18      }
19      return 1;
20  }
```

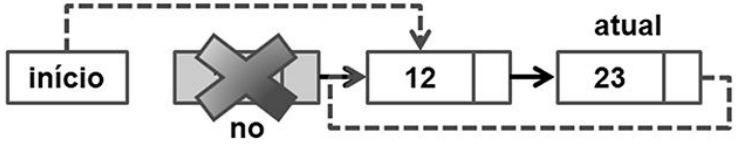
LISTA DINÂMICA ENCADEADA CIRCULAR - Removendo elementos

- Remover um elemento no final/início de uma **lista dinâmica encadeada circular** é uma tarefa um tanto trabalhosa:
  - Encontrar o último da lista e mudar os ponteiros para o início
  - É possível melhorar?



Busca último elemento: "atual"

```
atual = *li;
while(atual->prox != (*li))
   atual = atual->prox;
```

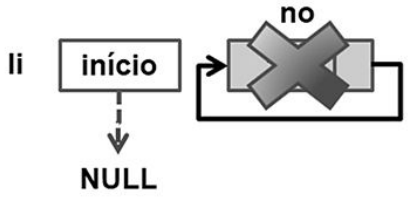


Remove o elemento

```
no = *li;
atual->prox = no->prox;
*li = no->prox;
free(no);
```

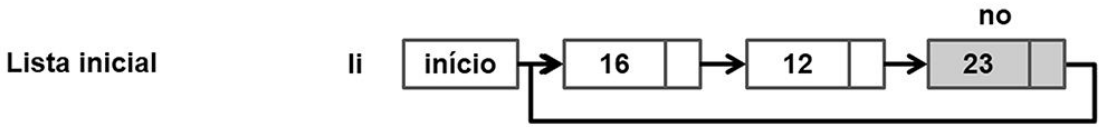
Se "no" é o único elemento da lista, a lista fica vazia:

```
free(*li);
*li = NULL;
```



LISTA DINÂMICA ENCADEADA CIRCULAR - Removendo elementos

- Remover um elemento no final/início de uma **lista dinâmica encadeada circular** é uma tarefa um tanto trabalhosa:
  - Encontrar o último da lista e mudar os ponteiros para o início
  - É possível melhorar?



Busca último elemento: "no"

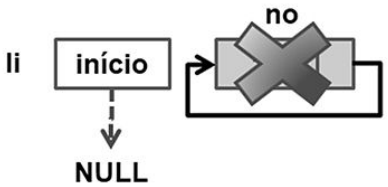
```
no = *li;
while(no->prox != (*li)){
   ant = no;
   no = no->prox;
}
```

Remove o elemento

```
ant->prox = no->prox;
free(no);
```

Se "no" é o único elemento da lista, a lista fica vazia:

```
free(*li);
*li = NULL;
```

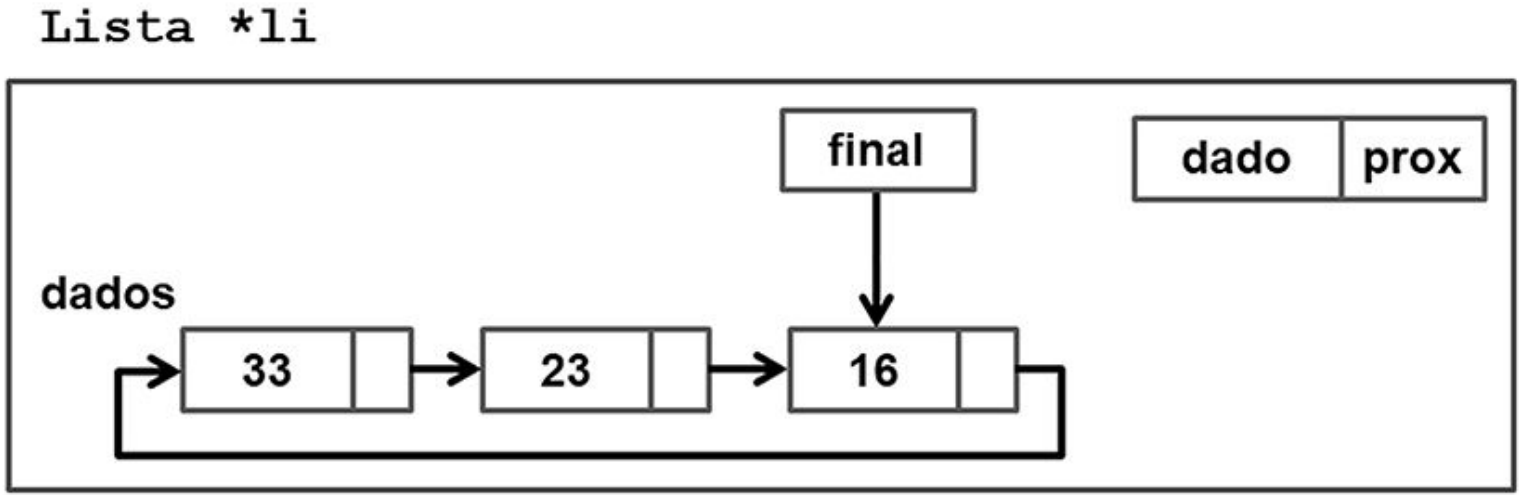


LISTA DINÂMICA ENCADEADA CIRCULAR - Aumentando o Desempenho

- Como visto até agora, as operações de inserção e remoção propostas anteriormente na **lista dinâmica encadeada circular** são bastante trabalhosas (custosas computacionalmente), principalmente quando realizadas no início ou no final da lista.
  - Como aumentar o desempenho (evitar ter que percorrer todos elementos)?

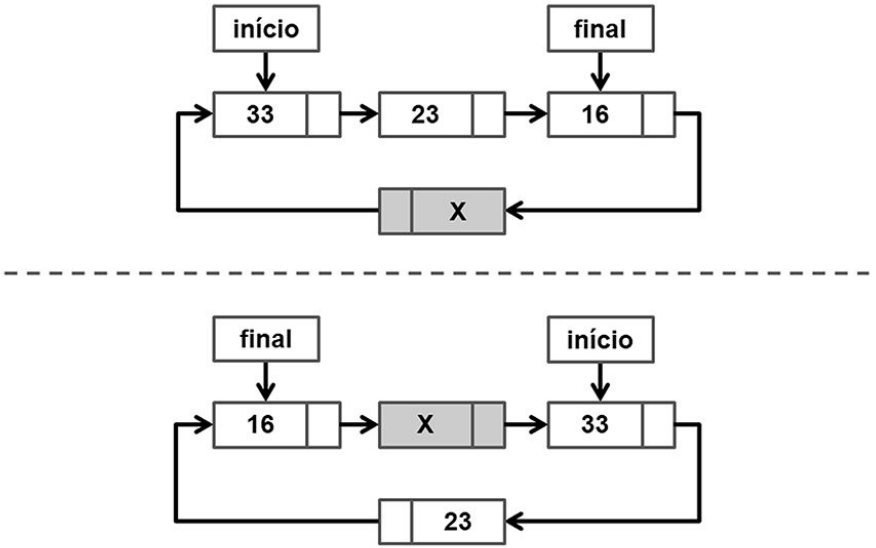
LISTA DINÂMICA ENCADEADA CIRCULAR - Aumentando o Desempenho

- Como visto até agora, as operações de inserção e remoção propostas anteriormente na **lista dinâmica encadeada circular** são bastante trabalhosas (custosas computacionalmente), principalmente quando realizadas no início ou no final da lista.
    - Para melhorar, uma opção é fazer com que a lista aponte para posição final ao invés da inicial:



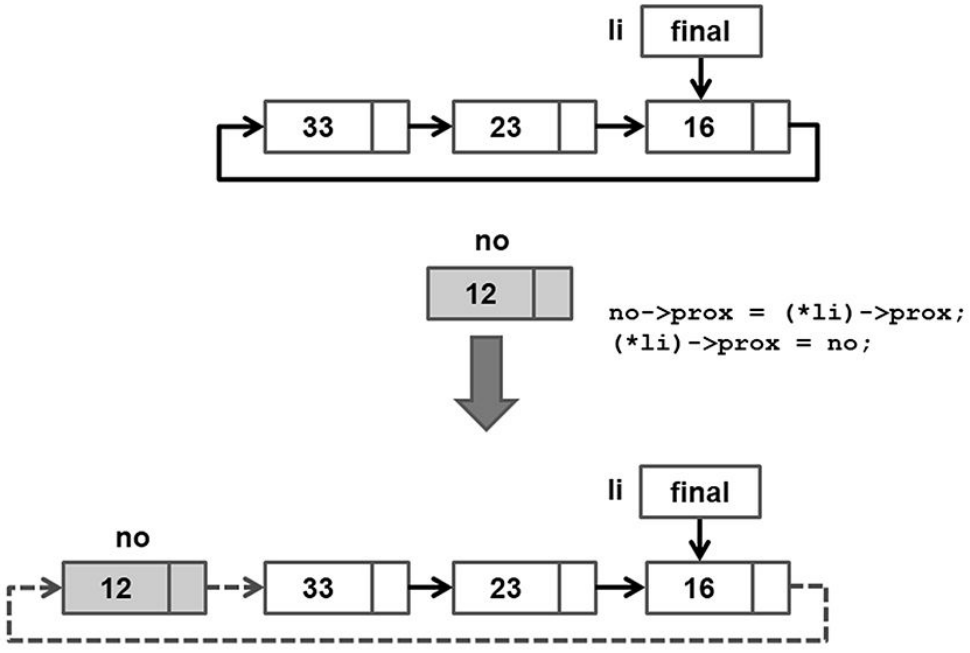
LISTA DINÂMICA ENCADEADA CIRCULAR - Aumentando o Desempenho

- Como visto até agora, as operações de inserção e remoção na **lista dinâmica encadeada circular** são bastante trabalhosas, principalmente quando realizadas no início ou no final da lista.
    - Imagine que queiramos inserir um novo elemento **X** em uma das extremidades da lista. Devido ao fato de a lista ser circular, inserir um novo elemento em qualquer uma de suas extremidades (início ou final) equivale a colocar esse novo elemento entre o seu **final** e o seu **início**. Assim, fazer com que a lista armazene o final dela não muda o funcionamento da lista, mas evita que se percorra a lista na inserção/remoção.



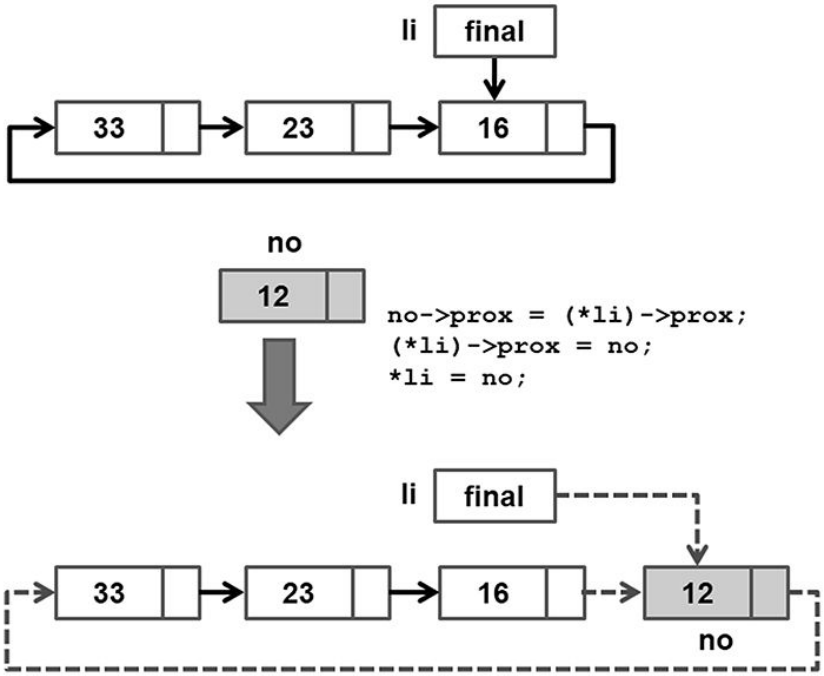
LISTA DINÂMICA ENCADEADA CIRCULAR - Aumentando o Desempenho

- Como visto até agora, as operações de inserção e remoção na **lista dinâmica encadeada circular** são bastante trabalhosas, principalmente quando realizadas no início ou no final da lista.
    - Para melhorar, basta apontar para posição final ao invés da inicial
    - Ex.: Inserindo no início:



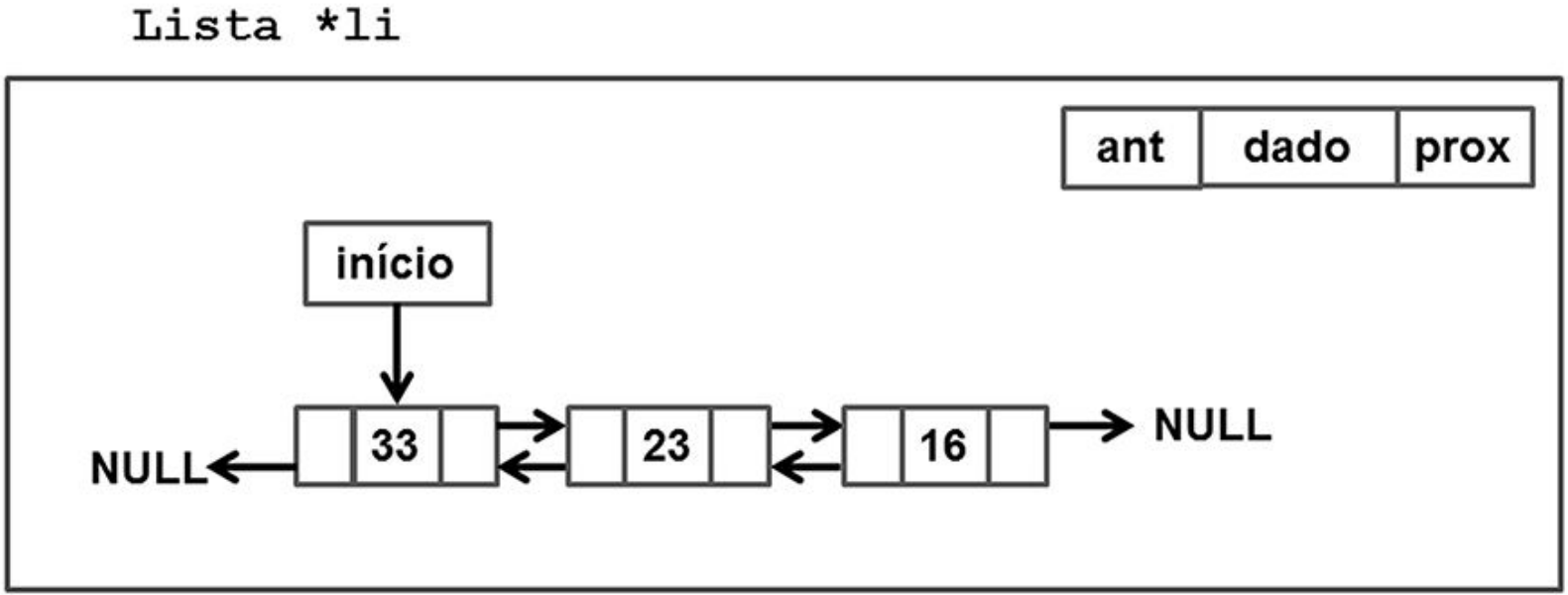
LISTA DINÂMICA ENCADEADA CIRCULAR - Aumentando o Desempenho

- Como visto até agora, as operações de inserção e remoção na **lista dinâmica encadeada circular** são bastante trabalhosas, principalmente quando realizadas no início ou no final da lista.
    - Para melhorar, basta apontar para posição final ao invés da inicial
    - Ex.: Inserindo no final:



LISTA DINÂMICA DUPLAMENTE ENCADEADA

- Diferente da **lista dinâmica encadeada**, esse tipo de lista não possui dois, mas sim **três** campos de informação dentro de cada elemento: os campos **dado**, **prox** e **ant**.
- A presença dos ponteiros **prox** e **ant** garantem que a lista seja encadeada em dois sentidos: no seu sentido normal, aquele usado para percorrer um lista do seu início até o seu final, e no sentido inverso, quando percorremos a lista de volta ao seu início.



LISTA DINÂMICA DUPLAMENTE ENCADEADA O que muda?

Arquivo ListaDinEncadDupla.h

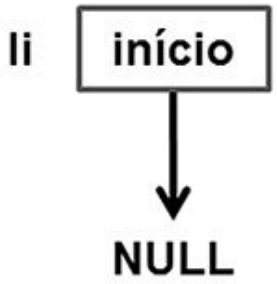
```
01  struct aluno{
02      int matricula;
03      char nome[30];
04      float n1,n2,n3;
05  };
06  typedef struct elemento* Lista;
07
08  Lista* cria_lista();
09  void libera_lista(Lista* li);
10  int busca_lista_pos(Lista* li, int pos, struct aluno *al);
11  int busca_lista_mat(Lista* li, int mat, struct aluno *al);
12  int insere_lista_final(Lista* li, struct aluno al);
13  int insere_lista_inicio(Lista* li, struct aluno al);
14  int insere_lista_ordenada(Lista* li, struct aluno al);
15  int remove_lista(Lista* li, int mat);
16  int remove_lista_inicio(Lista* li);
17  int remove_lista_final(Lista* li);
18  int tamanho_lista(Lista* li);
19  int lista_vazia(Lista* li);
20  int lista_cheia(Lista* li);
```

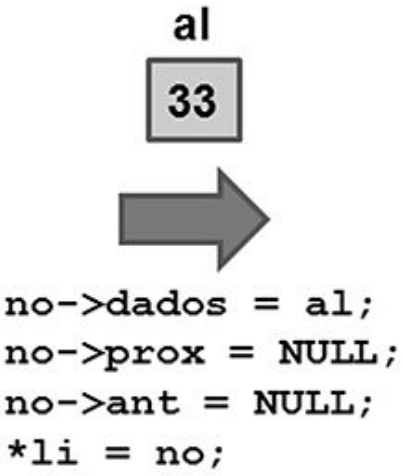
LISTA DINÂMICA DUPLAMENTE ENCADEADA O que muda?

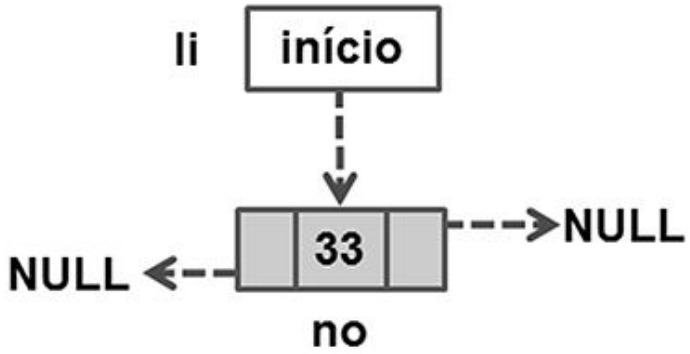
Arquivo ListaDinEncadDupla.c

```
01  #include <stdio.h>
02  #include <stdlib.h>
03  #include "ListaDinEncadDupla.h" //inclui os protótipos
04  //Definição do tipo lista
05  struct elemento{
06      struct elemento *ant;
07      struct aluno dados;
08      struct elemento *prox;
09  };
10  typedef struct elemento Elem;
```

LISTA DINÂMICA DUPLAMENTE ENCADEADA O que muda?





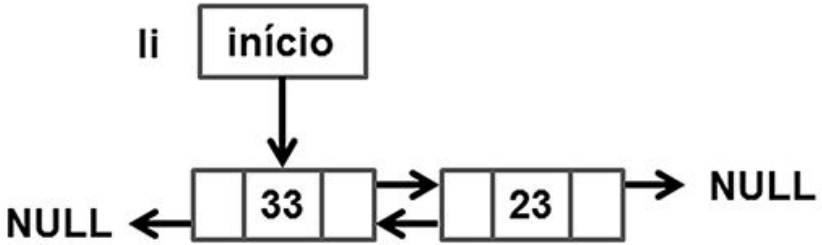


LISTA DINÂMICA DUPLAMENTE ENCADEADA O que muda?

Inserindo um elemento no início da lista

```
01  int insere_lista_inicio(Lista* li, struct aluno al){
02      if(li == NULL)
03          return 0;
04      Elem* no;
05      no = (Elem*) malloc(sizeof(Elem));
06      if(no == NULL)
07          return 0;
08      no->dados = al;
09      no->prox = (*li);
10      no->ant = NULL;
11      //lista não vazia: apontar para o anterior!
12      if(*li != NULL)
13          (*li)->ant = no;
14      *li = no;
15      return 1;
16  }
```

LISTA DINÂMICA DUPLAMENTE ENCADEADA O que muda?




```
no->dados = al;
no->prox = (*li);
no->ant = NULL;
```

al

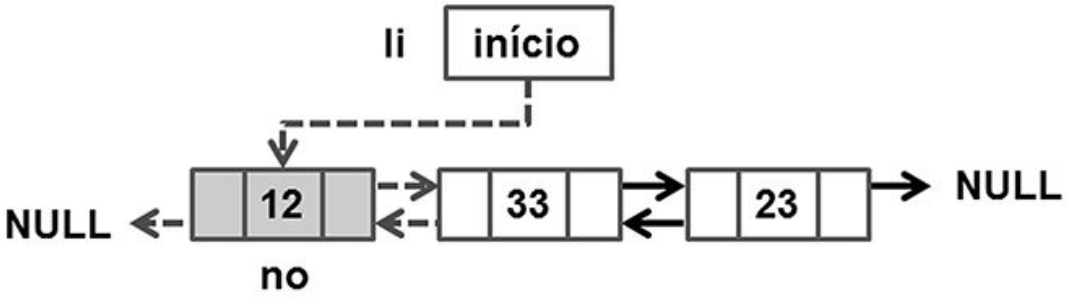
12

Lista não estava vazia:

```
(*li)->ant = no;
```

Por fim:

```
*li = no;
```



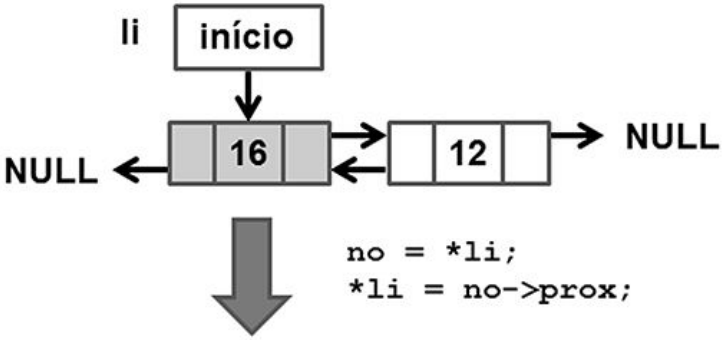
LISTA DINÂMICA DUPLAMENTE ENCADEADA O que muda?

Removendo um elemento do início da lista

```
01  int remove_lista_inicio(Lista* li){
02      if(li == NULL)
03          return 0;
04      if((*li) == NULL)//lista vazia
05          return 0;
06
07      Elem *no = *li;
08      *li = no->prox;
09      if(no->prox != NULL)
10          no->prox->ant = NULL;
11
12      free(no);
13      return 1;
14  }
```

LISTA DINÂMICA DUPLAMENTE ENCADEADA O que muda?

Lista inicial

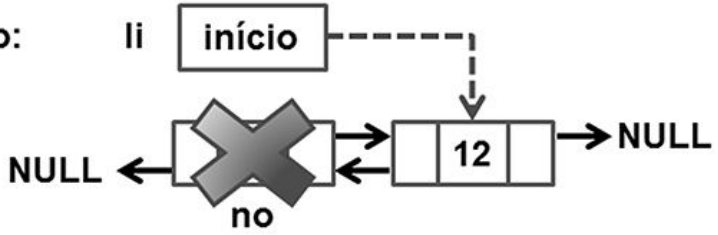


Se a lista possui mais de um elemento:

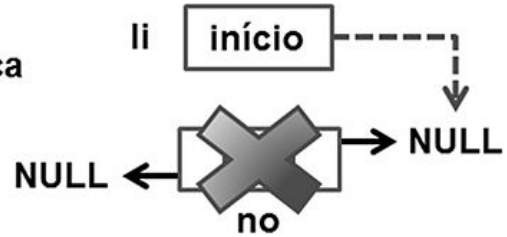
```
no->prox->ant = NULL;
```

Por fim:

```
free(no);
```



Se "no" é o único elemento, a lista fica vazia.



LISTA DINÂMICA DUPLAMENTE ENCADEADA O que muda?

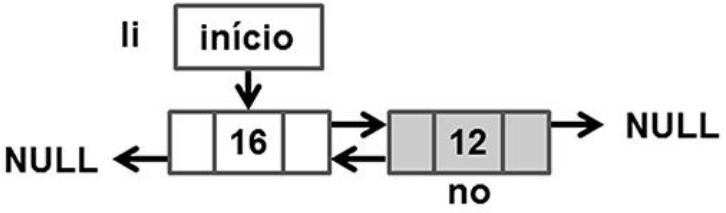
Removendo um elemento do final da lista

```
01  int remove_lista_final(Lista* li){
02      if(li == NULL)
03          return 0;
04      if((*li) == NULL)//lista vazia
05          return 0;
06
07      Elem *no = *li;
08      while(no->prox != NULL)
09          no = no->prox;
10
11      if(no->ant == NULL)//remover o primeiro e único
12          *li = no->prox;
13      else
14          no->ant->prox = NULL;
15
16      free(no);
17      return 1;
18  }
```

LISTA DINÂMICA DUPLAMENTE ENCADEADA O que muda?

Procura último elemento da lista:

```
no = *li;
while(no->prox != NULL)
    no = no->prox;
```

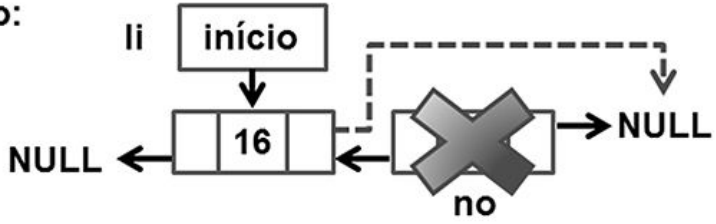


Se a lista possui mais de um elemento:

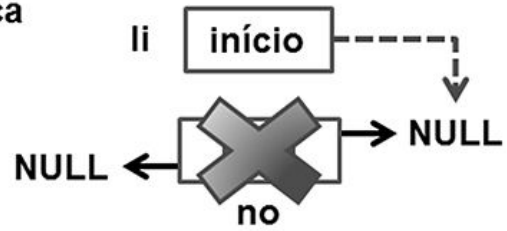
```
no->ant->prox = NULL;
```

Por fim:

```
free(no);
```



Se "no" é o único elemento, a lista fica vazia.



LISTA DINÂMICA DUPLAMENTE ENCADEADA O que muda?

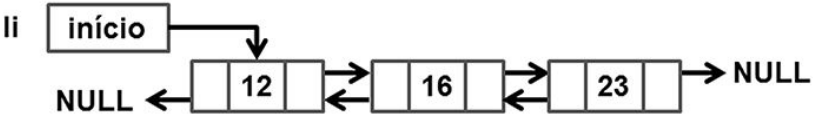
Removendo um elemento específico da lista

```
01  int remove_lista(Lista* li, int mat){
02      if(li == NULL)
03          return 0;
04      if((*li) == NULL)//lista vazia
05          return 0;
06      Elem *no = *li;
07      while(no != NULL && no->dados.matricula != mat){
08          no = no->prox;
09      }
10      if(no == NULL)//não encontrado
11          return 0;
12
13      if(no->ant == NULL)//remover o primeiro
14          *li = no->prox;
15      else
16          no->ant->prox = no->prox;
17
18      if(no->prox != NULL)//não é o último
19          no->prox->ant = no->ant;
20
21      free(no);
22      return 1;
23  }
```

LISTA DINÂMICA DUPLAMENTE ENCADEADA O que muda?

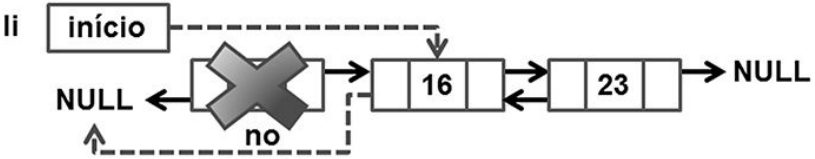
Lista inicial
Busca qual remover:

```
no = *li;
while(no != NULL && no->dados.matricula != mat){
   no = no->prox;
}
```



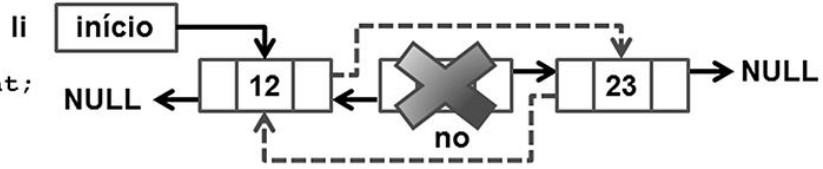
Remover do início:

```
*li = no->prox;
```



Não está removendo do final:

```
no->prox->ant = no->ant;
```

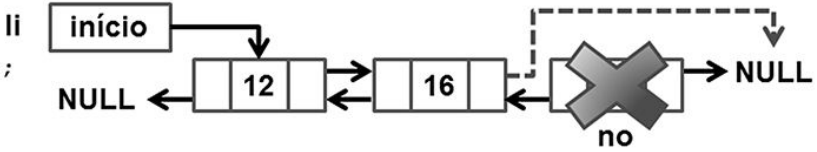


Remover do meio ou do final:

```
no->ant->prox=no->prox;
```

Por fim:

```
free(no);
```



LISTA DINÂMICA DUPLAMENTE ENCADEADA - Vantagens

- Não há necessidade de garantir um espaço mínimo para a execução da aplicação.
- Inserção e remoção em lista ordenada são as operações mais frequentes.
- Tamanho máximo da lista não é definido.
- Necessidade de acessar a informação de um elemento antecessor. (ex.: player de musica, exibição de lista de registros, páginas páginas da web, etc.)

Desvantagem:

- Dobro de memória com ponteiros.

Lista com Dinâmica Encadeada com Nó Descritor

- Além das informações do elemento, uma lista pode armazenar qualquer tipo de informação.
  - Para tanto, é necessário que especifiquemos isso na sua declaração.
  - Armazena informações específicas da lista (e não do nó)

Lista com Dinâmica Encadeada com Nó Descritor

Arquivo ListaDinEncadDesc.h

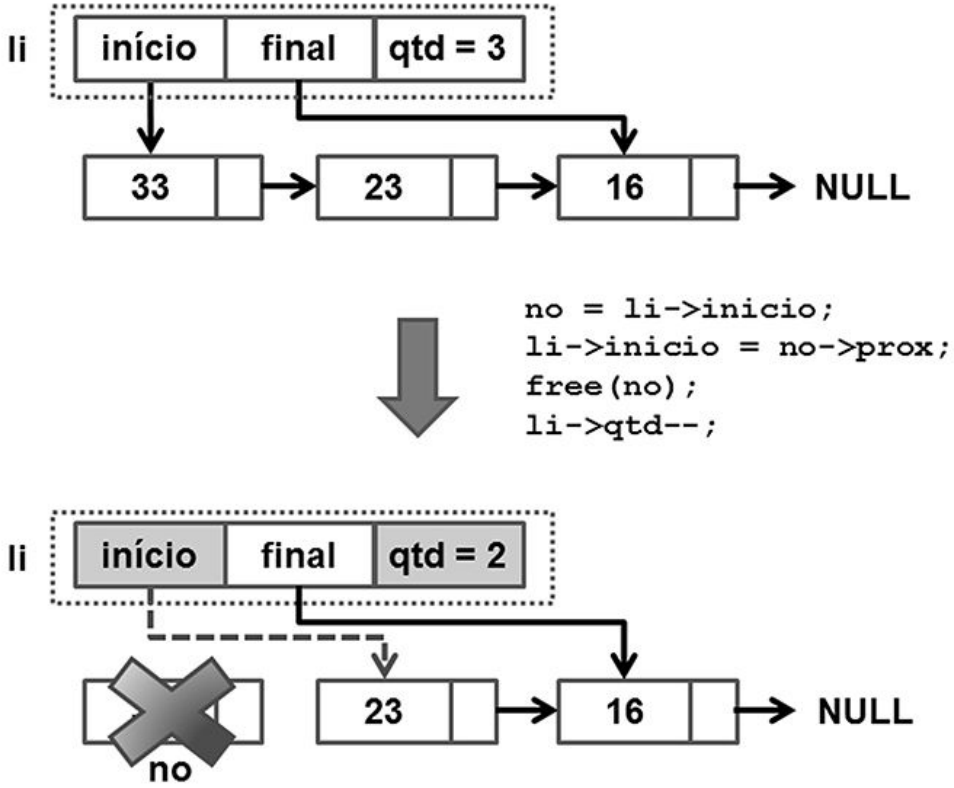
```
01  struct aluno{
02      int matricula;
03      char nome[30];
04      float n1,n2,n3;
05  };
06  typedef struct descritor Lista;
07
08  Lista* cria_lista();
09  void libera_lista(Lista* li);
10  int insere_lista_final(Lista* li, struct aluno al);
11  int insere_lista_inicio(Lista* li, struct aluno al);
12  int remove_lista_inicio(Lista* li);
13  int remove_lista_final(Lista* li);
14  int tamanho_lista(Lista* li);
15  int lista_vazia(Lista* li);
16  int lista_cheia(Lista* li);
17  int busca_lista_mat(Lista* li, int mat, struct aluno *al);
18  int busca_lista_pos(Lista* li, int pos, struct aluno *al);
```

Lista com Dinâmica Encadeada com Nó Descritor

Arquivo ListaDinEncadDesc.c

```
01  #include <stdio.h>
02  #include <stdlib.h>
03  #include "ListaDinEncadDesc.h" //inclui os protótipos
04  //Definição do tipo lista
05  struct elemento{
06      struct aluno dados;
07      struct elemento *prox;
08  };
09  typedef struct elemento Elem;
10
11  //Definição do Nó Descritor
12  struct descritor{
13      struct elemento *inicio;
14      struct elemento *final;
15      int tamanho;
16  };
```

Lista com Dinâmica Encadeada com Nó Descritor



Atividades (grupos de até 3)

1) Dada a lista dinâmica encadeada apresentada na aula (disponibilizada em https://www.facom.ufu.br/~backes/wordpress/ListaDinamicaEncadeada.zip), implemente uma função copia_lista que cria uma cópia de uma lista na memória de forma rápida (isso é, sem percorrer a lista de entrada múltiplas vezes). A função, definida abaixo, deve iterar por cada nó da lista original usando os ponteiros de próximo elemento, e retornar o ponteiro da nova lista:

```
Lista* copia_lista(Lista* li1)
```

Atividades (grupos de até 3)

2) Implementar um TAD referente à lista ordenada, usando alocação dinâmica simplesmente encadeada.

a) O nó dessa lista será descrito como:

```
struct no{
int info;
Struct no* prox;
};
typedef struct no Lista;
```

b) Operações do TAD da lista devem contemplar:

. Inicializar a lista
. Verificar se lista é vazia
. Inserir um dado elemento
. Remover um dado elemento
. Tamanho: retorna o número de elementos da lista
. Iguais: recebe duas listas ordenadas e verifica se elas são iguais
. Média: retorna a média aritmética simples dos elementos da lista
. Busca: verifica se um dado valor é pertencente à lista
. Elimina: elimina todas as ocorrências de um dado elemento

Atividades (grupos de até 3)

3) Dadas duas listas dinâmicas simplesmente encadeadas (com a mesma estrutura do item 2) com características: L1, L2, crie um algoritmo que irá gerar uma terceira lista com as seguintes

- Concatenar: recebe as listas L1 e L2 e retorna uma nova lista L3 formada pela concatenação entre a lista L1 e L2.
- Retornar ímpares: recebe as listas L1 e L2 e retorna uma nova lista L3, formada apenas com os elementos ímpares das duas listas.
- Conjunto intersecção: gerar uma terceira lista com a intersecção entre os números da primeira e da segunda lista.
- Intercalar: a função recebe duas listas ordenadas e retorna a lista com os elementos das duas listas intercalados conforme a ordem com que elas se dispõe na lista

https://docs.google.com/document/d/1bBi5aT9J2-m6CCJbOUQOnHQsAXSMoOQFFDQhlzW6eGw/edit?usp=sharing

Referências

Estrutura de Dados descomplicada em Linguagem C (André Backes): Cap 5;

Vídeo aulas (10-14):
https://programacaodescomplicada.wordpress.com/indice/estrutura-de-dados/

Implementações:

http://www.facom.ufu.br/~backes/wordpress/ListaDinamicaEncadeada.zip

http://www.facom.ufu.br/~backes/wordpress/ListaDinamicaEncadeadaCircular.zip

Projeto de Algoritmos (Nivio Ziviani): Capítulo 3;